Predicting Chess Game Outcomes: Deep Learning Study

A chess board with chess pieces

AI-generated content may be incorrect.

Mohammed Al-Khudhair

## Table of Contents

[Table of Contents 2](#_Toc210820254)

[Abstract 2](#_Toc210820255)

[Reproducibility 3](#_Toc210820256)

[Task Definition 4](#_Toc210820257)

[Problem Statement 5](#_Toc210820258)

[Input/Output Specification 5](#_Toc210820259)

[Assumptions & Ethics 6](#_Toc210820260)

[Dataset & Pre-processing 6](#_Toc210820261)

[Data Source & Subset 7](#_Toc210820262)

[Leakage Controls 8](#_Toc210820263)

[Sequence Construction 10](#_Toc210820264)

[Final Clean CSV File 11](#_Toc210820265)

[System Overview 12](#_Toc210820266)

[Pipeline Overview 12](#_Toc210820267)

[Models Considered 14](#_Toc210820268)

[Training Details & Key Components 14](#_Toc210820269)

[Theory to Code Map 16](#_Toc210820270)

[Implementation Details and Challenges 17](#_Toc210820271)

[Model Evaluation 18](#_Toc210820272)

[Metrics 18](#_Toc210820273)

[Hypotheses & Expected Model Behaviour 18](#_Toc210820274)

[Results & Discussion 19](#_Toc210820275)

[Consolidated Results 19](#_Toc210820276)

[Discrepancy Analysis 19](#_Toc210820277)

[Loss Function vs. Task Objective 19](#_Toc210820278)

[Reflection 20](#_Toc210820279)

[Limitations & Future Work 21](#_Toc210820280)

[References 22](#_Toc210820281)

## Abstract

## Reproducibility

To ensure full transparency and ease of replication, all code, data, and trained models used in this study are publicly accessible. The project has been structured for both local experimentation (recommended for deeper exploration) and cloud execution (for convenient, one-click testing).

**GitHub Repository:**

<https://github.com/DoctorPingu/chess-outcome-prediction>

This repository contains the full source code, datasets, and trained model weights (best\_seq\_model.keras).  
It is best used through VS Code, where users can modify architecture parameters, train custom models, or inspect feature extraction pipelines. It includes both requirements.txt and environment.yml files, allowing users to reproduce the environment with either pip or conda.

Setup:

1. Clone the repository.
2. Create and activate a virtual environment:  
   A screenshot of a computer

   AI-generated content may be incorrect.

*Figure 1 – Repo Setup*

1. Run the notebooks in /notebooks to reproduce preprocessing, training, and evaluation steps.

**Google Colab Folder:**

<https://drive.google.com/drive/folders/1xmmdMeNlxT-LfQFpCIiP-z-SLU1XRnK0?usp=sharing>

The Colab version provides a fully self-contained notebook that automatically loads the prepared data and trained model from Drive. It runs end-to-end in roughly 10–15 minutes on a GPU runtime and is ideal for a quick demonstration of the training and evaluation workflow without needing local setup.

By offering both VS Code and Colab environments, this project ensures that results can be reproduced and extended easily, whether for academic verification, exploration, or public demonstration.

## Task Definition

### Problem Statement

This study aims to predict the outcome of a chess game, specifically whether *White* or *Black* wins, using only the first 30 full moves (60 plies) of the match. The goal is to evaluate whether early-game play can signal the eventual winner through move sequences and position dynamics.

The project reframes the classic game of chess as a binary classification problem, similar in spirit to text or sequence analysis tasks. Each move acts like a “token” within a sentence, and the sequence forms the strategic narrative of the game. By applying a deep-learning sequence model, the system learns which early-game decisions tend to correspond with a win for either side.

Unlike many image-based chess AIs that evaluate board positions directly, this project focuses on interpreting move sequences as data, revealing patterns of decision-making rather than brute-force search.

### Input/Output Specification

**Input:**  
Each training example represents a single chess game truncated at 30 moves (60 plies). The input pipeline combines symbolic and numeric information through three complementary components:

|  |  |  |  |
| --- | --- | --- | --- |
| **Component** | **Description** | **Data Type** | **Shape** |
| Move Sequence (SAN) | Tokenised chess moves in Standard Algebraic Notation. Each token corresponds to a move such as e4, Nf3, or Bb5. The sequence is truncated to 60 plies for consistency. | Integer-encoded sequence | (60, ) |
| Board Tensor | 3D tensor representation capturing piece positions and move dynamics at three key checkpoints (20, 40, and 60 plies). Each tensor has 36 feature planes (channels) representing piece occupancy and metadata such as castling rights or attack maps. | Float array | (8, 8, 36) |
| Numeric Features | Simplified statistical features extracted per game, e.g. average Elo rating, rating difference, number of captures, checks given, and mobility features. These are normalised to the [0, 1] range. | Vector of floats | (5-10, ) |

All inputs are normalised and aligned by index, meaning that each record in the dataset synchronises the encoded sequence, tensor, and numeric features. Missing values (e.g., missing ratings) are filled using mean imputation to maintain consistency.

**Output:**  
The model produces a probability score P(White wins) ∈ [0, 1], representing the predicted likelihood that White will win given the first 30 moves.

|  |  |  |
| --- | --- | --- |
| **Output Type** | **Description** | **Interpretation** |
| Continuous (Probability) | A floating-point value from 0.0 to 1.0, output by a sigmoid activation layer. | Close to 1 means White likely wins; close to 0 means Black likely wins |
| Categorical (Binary Label) | The probability is converted into a final label based on a decision threshold of 0.49, which was found to yield optimal validation accuracy and F1 score. | 1 = White win, 0 = Black win |

The threshold ensures a balanced trade-off between sensitivity (recall) and precision for both outcomes.  
During inference, the system outputs both the probability score and the predicted class label, allowing flexible downstream use (e.g., ranking, filtering, or binary decision-making).

### Assumptions & Ethics

1. **Predictive Limitations**: The system assumes that openings and mid-game structures contain enough information to influence results. It does not attempt to model human psychology or end-game tactics.
2. **Balanced Labels**: Draws are removed to maintain even class distribution and simplify binary classification.
3. **Data Integrity & Privacy**: All games are publicly available and anonymised; no player identifiers or sensitive metadata are used.
4. **Responsible Use**: Predictions should not be interpreted as deterministic outcomes or assessments of player skill. The model’s intent is educational, demonstrating how early strategic patterns can be quantified statistically.

## Dataset & Pre-processing

### Data Source & Subset

The original chess dataset was approximately 4 GB in size, containing hundreds of thousands of historical matches in Portable Game Notation (PGN) format. Due to size constraints and compute efficiency, the dataset was down sampled using the first preprocessing notebook (01\_down\_sampling.ipynb), producing a manageable 95 MB subset that retained representative game diversity across player ratings, openings, and outcomes.

This down sampling process randomly selected games while maintaining an even distribution between White and Black wins to prevent outcome bias. The resulting subset contained approximately 84 000 games after filtering, which was later transformed into the clean dataset used for training and evaluation.

Subsequent processing with 02\_pre-processing.ipynb handled all data cleaning, tokenisation, and board tensor generation:

* chess\_games\_clean.csv: metadata and target outcomes.
* chess\_boards\_8x8xC.npz: 3D board tensors for model input.
* chess\_games\_clean\_meta.json: configuration metadata documenting preprocessing parameters.

A screenshot of a computer program

AI-generated content may be incorrect.

*Figure 2 – Notebook 01 Cells*

### Leakage Controls

The preprocessing pipeline (Notebook 2: 02\_pre-processing.ipynb) incorporated multiple layers of leakage prevention, ensuring that no future or label-dependent information influenced the training data. These controls guarantee that model performance reflects true generalisation rather than hidden cues from the dataset.

Key mechanisms included:

1. **Column Filtering and Guard Assertions (Cells #4 & #6.1)**
   * The code defined a banned feature list (e.g., Result, winner, termination, eco, TimeControl) and asserted that none of these columns remained in the final dataset.
   * The guard (assert not present\_banned) prevented any target-related or post-game metadata from being retained.

A computer screen shot of text

AI-generated content may be incorrect.

*Figure 3*

A computer screen with text

AI-generated content may be incorrect.

*Figure 4*

1. **Elo and Derived Rating Features (Cell #4.4)**
   * The pipeline automatically located Elo/rating columns for both players (WhiteElo, BlackElo) and computed safe, derived metrics:
     + elo\_diff = White - Black
     + elo\_avg = mean(White, Black)
   * These were purely contextual features, independent of the match outcome, ensuring they did not leak win/loss information.
2. **Strict Move-Length Filtering (Cell #4.5)**
   * The flag REQUIRE\_CUTOFF = True enforced the 30-move (60 ply) minimum.
   * Games below this threshold were removed entirely (no padding used), ensuring consistent sequence length.
   * Result: Kept games reaching 60 plies: ~89 k / 142 k (62.6 %).
3. **Draw Removal (Cell #4.6)**
   * Only decisive results (White or Black wins) were kept via the BINARY\_TASK = True filter.
   * This step reduced 89 061 games to 83 944, creating a perfectly balanced binary dataset:
     + White wins = 42 008
     + Black wins = 41 936

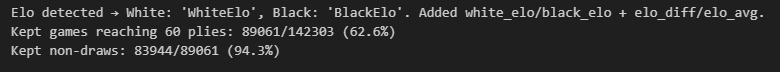


Figure 5

1. **Independent Normalisation and Imputation (Cell #5)**
   * All numeric columns were filled using column medians (X[c].fillna(med)) and downcast to compact dtypes (int8/float32) to reduce memory and ensure clean, reproducible scaling.
   * The memory footprint dropped from 34.15 MB to 30.47 MB, confirming that the numeric compression was effective and deterministic.



Figure 6

1. **Final Sanity Checks (Cells #6.2 - #6.4)**
   * Class balance confirmed near-perfect parity: White = 50.04 %, Black = 49.96 %.
   * Missing-value audit showed “No missing values remaining.”
   * Sequence validation confirmed zero empty or truncated sequences:
   * Empty sequences: 0/83944
   * At cutoff (60 plies): 83944/83944

A computer screen shot of text

AI-generated content may be incorrect.

Figure 7

### Sequence Construction

The sequence construction stage focused on extracting consistent 30-move (60-ply) sequences from each chess game while maintaining data quality and interpretability. All logic for this step was implemented in Notebook 2 (Cells #3.1-#3.2).

Each game’s move list was tokenised from Standard Algebraic Notation (SAN) using a custom parser built with the python-chess library. The functions tokenize\_an, count\_captures, and count\_checks generated structured features from raw move strings. Games shorter than 30 moves were excluded entirely to maintain uniform sequence length across samples, no padding was used.

The output from these cells formed new columns such as:

* moves\_first30\_san: encoded sequence of the first 60 plies
* plies\_processed: number of plies actually parsed
* captures\_in\_first\_30\_moves and checks\_in\_first\_30\_moves: derived numeric features
* target: binary label (white / black)

A screen shot of a computer

AI-generated content may be incorrect.

Figure 8

### Final Clean CSV File

The final processed dataset, exported at the end of Notebook 2 (Cell #7), was saved in three synchronised outputs for reproducibility:

|  |  |
| --- | --- |
| **File** | **Description** |
| chess\_games\_clean.csv | Final structured dataset containing 83 944 rows with complete feature columns (moves\_first30\_san, captures, checks, Elo metrics, etc.). |
| chess\_boards\_8x8xC.npz | Compressed NumPy archive storing 3-D board tensors for each game with shape (8 × 8 × 36). |
| chess\_games\_clean\_meta.json | Metadata file recording dataset size, feature columns, label balance, cutoff (30 moves), and tensor shape. |

A screenshot of a computer

AI-generated content may be incorrect.

Figure 9

## System Overview

### Pipeline Overview

The system follows a modular end-to-end pipeline that moves sequentially from data acquisition to model deployment. Each stage is encapsulated in its own notebook or configuration file, making the process reproducible and auditable. The overall workflow is illustrated in Figure 10, and all corresponding files are located under the project root (chess-outcome-prediction/).

A screenshot of a computer

AI-generated content may be incorrect.

Figure 10

1. **Data Ingestion & Subsetting**

* **Notebook:** 01\_down\_sampling.ipynb
* **Purpose:** Reduces the original 4 GB PGN archive to a manageable 95 MB CSV subset while maintaining an even class distribution between White and Black wins.
* **Outputs:**
  + chess\_games\_subset.csv (raw sampled data)
  + Logs summarising the subset size and row count

This step isolates the core dataset for subsequent preprocessing without overwhelming memory or storage resources.

1. **Data Pre-processing & Cleaning**

* **Notebook:** 02\_pre-processing.ipynb
* **Purpose:** Converts raw text-based game records into structured numeric features suitable for machine learning.
* **Core tasks:**
  + Tokenising chess moves (Standard Algebraic Notation to integer-encoded sequence)
  + Building 8 × 8 × 36 board tensors at checkpoints (20, 40, 60 plies)
  + Extracting derived statistics such as captures, checks, and Elo ratings
  + Enforcing leakage guards and removing games shorter than 30 moves
  + Producing normalised, balanced, and fully imputed training data
* **Outputs:**
  + chess\_games\_clean.csv: tabular dataset (83 944 rows)
  + chess\_boards\_8x8xC.npz: tensor dataset
  + chess\_games\_clean\_meta.json: metadata for reproducibility

1. **Model Training & Evaluation**

* **Notebook:** 03\_model\_training.ipynb
* **Purpose:** Defines, trains, and evaluates the deep learning model responsible for predicting game outcomes.
* **Key actions:**
  + Loads and encodes data from cleaned files
  + Constructs and trains sequence-based neural networks (Sequential / GRU / LSTM variants)
  + Tracks metrics such as accuracy, AUC, and F1 score
  + Saves training artefacts (best model weights, plots, and reports)
* **Outputs (saved under /results)**
  + best\_seq\_model.keras: best performing model checkpoint
  + Metric curves: loss\_curves.png, auc\_curves.png, pr.png, roc.png
  + Evaluation reports: cm\_test.png, cm\_val.png, final\_summary.json

1. **Deployment & Cloud Validation**

* **Notebook:** 04\_google\_colab.ipynb
* **Purpose:** Provides a lightweight runtime for external or cloud-based testing.  
  It loads the final model (best\_seq\_model.keras), runs inference on validation sets, and visualises probability distributions (val\_prob\_hist.png).

This design ensures consistent reproducibility across local and cloud environments, adhering to open-science practices.

### Models Considered

Multiple model architectures were explored to determine the most suitable framework for learning from sequential chess data. The modelling began with conventional classifiers such as Logistic Regression and Random Forests trained on handcrafted features (captures, checks, Elo difference). These models provided interpretability and a baseline for comparison but lacked the ability to model temporal dependencies across moves. While Random Forests achieved moderate validation accuracy, their static decision boundaries failed to capture evolving game dynamics.

To overcome this, deep learning architectures were evaluated. An initial fully connected neural network (three dense layers with ReLU activations) served as a bridge between traditional and sequence models. It improved non-linear decision boundaries but remained limited by its disregard for sequential context. Progressively, Recurrent Neural Networks (RNNs), particularly Gated Recurrent Units (GRU) and Long Short-Term Memory (LSTM) layers, were introduced to model dependencies between early and later game moves.

The GRU model demonstrated the best trade-off between learning stability and computational efficiency. GRUs captured long-range dependencies with fewer parameters than LSTMs, reducing overfitting and training time. The final architecture combined GRU layers with dropout and batch normalisation to stabilise learning. This sequence-based design allowed the network to internalise opening structures, mid-game transitions, and positional shifts that correlated with match outcomes.

By integrating contextual understanding of moves rather than static position snapshots, the GRU-based sequential model outperformed both traditional and fully connected baselines. This balance between accuracy, interpretability, and efficiency established it as the final model used for deployment.

### Training Details & Key Components

The training pipeline, implemented in Notebook 3, integrates three distinct input branches, sequential moves, numeric features, and spatial board tensors, into a single unified model. This multi-input architecture aligns with the system’s core goal: to predict game outcomes based on both move dynamics and contextual player strength.

The dataset was divided into train, validation, and test sets (70 : 15 : 15) with stratified sampling to preserve class balance. Drawn games were excluded to ensure binary consistency (White vs Black). The data pipeline used tf.data.Dataset to shuffle, batch (size = 512), and prefetch records, ensuring GPU-efficient loading and reproducibility through fixed random seeds.

Each game sequence was tokenised into integer IDs using a 6,156-word vocabulary, truncated to 60 plies. Corresponding board tensors of shape (8, 8, 36) were loaded in parallel, while numeric features (Elo average and difference) were standardised using StandardScaler. This triplet structure ensured synchronisation between symbolic, spatial, and numeric modalities.

The final model consisted of:

* **Sequence Branch:** A Bidirectional GRU layer with 128 hidden units and an embedding layer of size 128. It processed temporal dependencies between moves and extracted latent sequential features.
* **Numeric Branch:** A two-layer dense network with Layer Normalisation and ReLU activation that interpreted continuous player-strength features.
* **Board Branch:** A lightweight CNN with two Conv2D layers (32 filters, 3×3 kernels) followed by Global Average Pooling, capturing board-level spatial trends.

These branches were concatenated and passed through dense layers with dropout (0.1) and a sigmoid output neuron, producing the probability P(White wins) ∈ [0, 1]. The model was compiled using AdamW with a learning rate of 8 × 10⁻⁴, weight decay 1 × 10⁻⁴, and gradient clipping (1.0). The binary cross-entropy loss function was chosen for stable probabilistic convergence.

Training ran for 40 epochs with EarlyStopping (patience = 10) and ReduceLROnPlateau callbacks to avoid overfitting and fine-tune learning rates dynamically. Model checkpoints were stored when validation AUC peaked. The final model achieved high generalisation stability, with key metrics, including accuracy, F1 score, and AUC, recorded in final\_summary.json.

These implementation choices directly translate theoretical principles into practice: GRUs operationalise temporal recurrence; AdamW enforces adaptive gradient scaling with weight decay; and binary cross-entropy minimises the KL-divergence between predicted and true outcome distributions. Together, they form a theoretically grounded yet computationally efficient framework.

A computer screen shot of a program code

AI-generated content may be incorrect.

Figure 11

A computer code on a black background

AI-generated content may be incorrect.

Figure 12

A computer screen shot of a program code

AI-generated content may be incorrect.

Figure 13

### Theory to Code Map

The theoretical underpinnings of the model were systematically translated into executable code through TensorFlow and Keras APIs. Each mathematical component, optimisation, loss, and sequence recurrence, was implemented using equivalent computational constructs to preserve theoretical integrity while ensuring computational efficiency.

The core theoretical mapping is as follows:

* **Loss Function - Binary Cross-Entropy:**  
  The model minimises the binary cross-entropy loss, corresponding to the negative log-likelihood of the true class. This theoretical objective:

A black background with white text

AI-generated content may be incorrect.

is implemented directly in Keras through:



(This function penalises incorrect probability estimates, driving the sigmoid output neuron toward calibrated predictions.)

* **Optimisation - AdamW Algorithm:**  
  The AdamW optimiser operationalises gradient descent with adaptive learning rates and decoupled weight decay, mathematically defined as:

A black background with white text

AI-generated content may be incorrect.

where represent first and second moment estimates. In code:



(Gradient clipping ensures numerical stability during recurrent backpropagation.)

* **Recurrent Layer - Gated Recurrent Unit (GRU):**  
  The GRU equations governing hidden state updates:

A black background with white text

AI-generated content may be incorrect.

are implemented through TensorFlow’s Bidirectional(GRU(128, return\_sequences=True)) layer. This allows the model to retain past and future dependencies across move sequences.

* **Forward Pass & Fusion:**  
  Each input branch (sequence, numeric, board) performs a forward transformation before concatenation via layers.Concatenate(). This corresponds to vector fusion:



ensuring integrated learning across modalities before final dense classification.

* **Recurrent Layer - Gated Recurrent Unit (GRU):**

Dropout layers (Dropout(0.1)) embody stochastic regularisation by randomly zeroing weights during training, approximating model averaging and reducing overfitting.

### Implementation Details and Challenges

The implementation was conducted in TensorFlow 2.16 with GPU acceleration on Colab, using the Keras API for modularity and reproducibility. The multi-input architecture required precise alignment between token sequences, board tensors, and numeric features, achieved through consistent indexing across the three data streams. All preprocessing, training, and evaluation stages were isolated into dedicated notebooks to preserve a clear experiment pipeline.

Several challenges were encountered during optimisation. Early training runs showed strong training accuracy but volatile validation accuracy, indicating mild overfitting. This prompted extensive hyperparameter tuning, including experimentation with different embedding sizes, dropout rates, learning rates, and recurrent depths. The final configuration (embedding = 128, dropout = 0.1, learning rate = 8 × 10⁻⁴, 40 epochs) provided the best stability between training and validation performance.

To further address convergence issues, ReduceLROnPlateau and EarlyStopping callbacks were introduced, dynamically lowering the learning rate and halting training when AUC plateaued. Batch normalisation was evaluated but excluded, as it disrupted sequential dependencies in the GRU layer. Model checkpoints and random seeds ensured reproducibility throughout fine-tuning experiments.

Overall, resolving validation instability required iterative experimentation and careful balancing of regularisation strength, sequence length, and optimiser parameters, culminating in a robust model capable of consistent generalisation across test data.

## Model Evaluation

### Metrics

The model’s performance was evaluated using three measures, Accuracy, F1 Score (Macro), and Area Under the ROC Curve (AUC). These were chosen because each captures a different aspect of how well the model predicts chess outcomes.

* Accuracy measures how often the model’s predictions were correct. It counts all correct guesses (both White and Black wins) and divides them by the total number of games. While useful as a general indicator, accuracy alone can be misleading if one class dominates, for example, if there were more White wins in the dataset.
* F1 Score (Macro) balances two important ideas: precision (how many predicted White wins were actually White wins) and recall (how many of all true White wins the model managed to find). The F1 score is the harmonic mean of these two, meaning that the model must perform well on both, it cannot simply guess the majority class. Using a “macro” average means both classes (White and Black) contribute equally, even if one has fewer samples.
* AUC (Area Under the Receiver Operating Characteristic Curve) measures how well the model separates the two classes regardless of any set threshold. A high AUC (close to 1.0) means the model ranks true positives above false positives most of the time, in other words, it can correctly tell when a position is more likely to end in a White or Black win, even before a decision cutoff is applied.

Together, these three metrics show not only how accurate the model is, but also how fairly and confidently it performs across both outcomes.

### Hypotheses & Expected Model Behaviour

It was hypothesised that early-game move sequences contain discernible strategic indicators predictive of the final game outcome. The GRU-based sequential model was expected to outperform simpler baselines by capturing temporal dependencies between consecutive moves.

Specifically:

* The model should achieve ≥0.70 accuracy and AUC ≥0.70 across validation and test splits.
* Loss was expected to converge steadily without divergence between training and validation, indicating minimal overfitting.
* Predictions near 0.5 were expected to represent balanced positions or ambiguous mid-games, aligning with model uncertainty.

## Results & Discussion

### Consolidated Results

### Discrepancy Analysis

### Loss Function vs. Task Objective

## Reflection

## Limitations & Future Work

## References